Automatic Generation of Software Pipelines for Heterogeneous Parallel Systems

Jacques A. Pienaar†, Srimat Chakradhar‡, and Anand Raghunathan†
†School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN
‡NEC Laboratories America, Princeton, NJ
{jpienaar,raghunathan}@purdue.edu, chak@nec-labs.com

***Abstract*—Pipelining is a well-known approach to increasing parallelism and performance. We address the problem of software pipelining for heterogeneous parallel platforms that consist of different multi-core and many-core processing units. In this context, pipelining involves two key steps—partitioning an application into stages and mapping and scheduling the stages onto the processing units of the heterogeneous platform. We show that the inter-dependency between these steps is a critical challenge that must be addressed in order to achieve high performance. We propose an Automatic Heterogeneous Pipelining framework (AHP) that generates an optimized pipelined implementation of a program from an annotated unpipelined specification. Across three complex applications (image classification, object detection, and document retrieval) and two heterogeneous platforms (Intel Xeon multi-core CPUs with Intel MIC and NVIDIA GPGPU accelerators), AHP achieves a throughput improvement of up to 1.53x (1.37x on average) over a heterogeneous baseline that exploits data and task parallelism.**

I. Introduction

Heterogeneity has emerged as a prevalent characteristic of parallel computing platforms. Heterogeneous parallel platforms combine processing units (PUs) that have different instruction set architectures or different micro-architectures, *e.g.*, multi-core processors, general-purpose graphics processing units (GPGPUs) [1] and other many-core accelerators [2], [3]. The PUs that constitute a heterogeneous platform are effectively optimized to serve different workload characteristics (*e.g.*, latency-sensitive *vs.* throughput-sensitive, coarse-grained *vs.* fine-grained parallel, *etc.*). Studies have argued for the fundamental benefits of heterogeneity in parallel computing [4] and demonstrated speedups on heterogeneous platforms for a wide range of application domains [5]. However, programming heterogeneous parallel platforms is commonly cited as a major challenge that must be addressed before their potential can be realized for mainstream computing.

Progress has been made in raising the level of abstraction at which many-core accelerators GPUs are programmed—from low-level, domain-specific APIs to high-level programming languages [6]–[9]. While they cover much ground, these frameworks still leave the programmer with the significant challenges of tuning the accelerator code for performance, partitioning, mapping and scheduling an application on the different PUs of a heterogeneous platform, and managing the data transfers between their (often distinct) memory hierarchies. Recent research efforts address the challenge of providing the programmer with a unified view of the ensemble of PUs and their memory hierarchies on the one hand, while on the other hand achieving good performance and performance portability [10]–[21].

While addressing heterogeneity in the computing platform, it is also important to recognize the heterogeneity in how parallelism manifests in applications [22]. The heterogeneous programming frameworks described above have extensively focused on data and task parallelism. Pipeline parallelism is another important form of parallelism that is complementary to task and data parallelism. Previous research has shown that it is important to consider all three forms of parallelism and exploit the right combination thereof to get the best performance on a given parallel platform [23], [24]. Yet, little work exists that exploits pipeline parallelism in heterogeneous parallel platforms. Our work aims to fill this gap, by providing a framework for the generation of optimized heterogeneous software pipelines from unpipelined specifications.

Application programming interfaces (APIs) for specification of pipeline patterns are provided in modern parallel programming frameworks [25], [26]. These frameworks require the programmer to manually identify the pipeline stages and program them using the provided API. However, identifying the best partition of a program into pipeline stages can be quite complex. Automatic software pipelining has a rich history and has been explored in various contexts such as loop pipelining in VLIW compilers [27], [28]. Our context, in which the execution of the program is pipelined at a coarse granularity across PUs, differs significantly from these efforts. For homogeneous multi-core platforms, the problem of partitioning a program into pipeline stages has been extensively addressed [29]–[39]. While some of these efforts have targeted accelerators [37], [39], the stages of the pipeline execute entirely on the (homogeneous) cores of the accelerator, therefore the problem is conceptually the same as the homogeneous case.

For heterogeneous platforms, the problem of software pipelining becomes significantly more challenging. The execution time of a stage is highly sensitive to how the computations in the stage are mapped and scheduled on the PUs in the heterogeneous platform. On the other hand, the mapping and scheduling also depend on how the application is delineated into pipeline stages, since stages execute concurrently, affecting the contention for PUs. *The two key steps of pipeline stage identification, and pipeline mapping and scheduling need to be performed considering this strong inter-dependency.*

Our work makes the following contributions:

- We identify the unique challenge involved in software pipelining for heterogeneous parallel platforms, *viz.* the inter-dependency between partitioning the program into pipeline stages, and mapping and scheduling the stages onto the processing units.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

- We propose AHP, an automatic framework for the creation of heterogeneous pipelines from non-pipelined program specifications. Given an annotated program, AHP extracts a task graph from each target program region, profiles the tasks on each of the processing units in the platform, and iteratively partitions the task graph into pipeline stages and schedules the pipelined task graph onto the heterogeneous platform. This iterative process accounts for the inter-dependency mentioned above.
- We evaluate AHP by using it to generate pipelined implementations of three complex applications—namely 1) image classification, 2) object detection using convolutional neural networks, and 3) document retrieval using supervised semantic indexing—on two heterogeneous platforms, one with an Intel Xeon multi-core and an NVIDIA Tesla GPGPU, and another with an Intel Xeon multi-core and an Intel MIC accelerator. Our results demonstrate (i) the importance of exploiting pipeline parallelism on heterogeneous platforms, and (ii) the need to consider the inter-dependency between pipeline stage identification and mapping and scheduling in order to maximize performance.

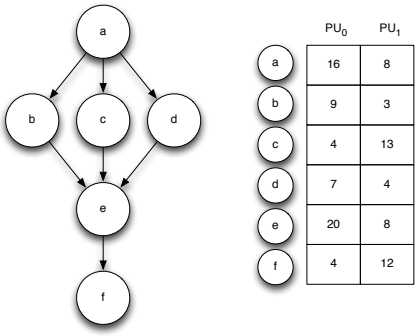
II. Background and Motivation

In this section, we describe the setting for our work, and illustrate the key challenge involved in software pipelining for heterogeneous parallel platforms.

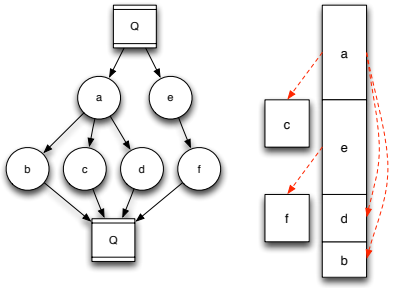
We assume that the program region to be pipelined can be represented as a task graph, which consists of tasks and dependencies between them. Several contemporary parallel programming frameworks adopt a similar approach to represent parallelism [14], [15], [40], [41]. The task graph representation encapsulates all three forms of parallelism as each task may contain internal parallelism (typically data parallelism), task parallelism is present across tasks in the task graph, and different instances of the task graph processing different inputs may be executed in an overlapping manner, pipelining its execution.

The processing platform consists of distinct PUs, such as multi-core processors, GPGPUs, and other many-core accelerators such as Intel's MIC [3] or IBM's Cell [2]. Each of the tasks in the task graph may be executed on a subset of the PUs. We do not address the problem of generating optimized implementations of individual tasks for PUs—we assume the availability of compiler frameworks or hand-optimized implementations of tasks. The implementation of a task may exploit parallelism internal to a PU (*e.g.*, the cores within a GPU or MIC), and task parallelism is exploited by scheduling independent tasks to execute concurrently [11], [12], [18], [21]. In this work, we are concerned with the pipelined execution of the task graph across the PUs. Any sub-graph of the task graph may form a pipeline stage, although achieving good performance requires that these stages be identified judiciously.

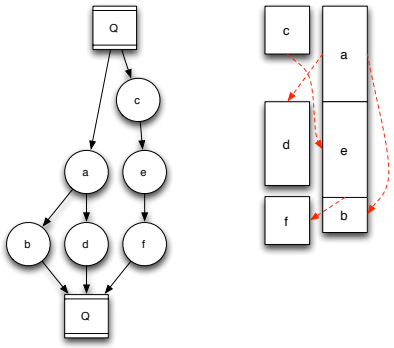
Within the context described above, exploiting pipeline parallelism efficiently requires us to create pipeline stages from the task graph, and mapping and scheduling the pipelined task graph onto the parallel platform. *A key motivation for our work is that, for heterogeneous platforms, these two steps are highly inter-dependent, and ignoring this strong cyclic dependency leads to sub-optimal performance*. We qualitatively explain this key insight and illustrate it using an example (we empirically show the effect in Section IV).



(a) Task graph with execution times



(b) Pipeline 1 (23 time units)



(c) Pipeline 2 (20 time units)

Fig. 1. Example task graph with alternative pipeline structures and schedules.

It is well known that a pipeline's throughput is limited by its longest running stage. Hence, a pipeline with all its stages running nearly the same duration, *i.e.*, a balanced pipeline, is desirable as it provides the best throughput. To create a balanced pipeline one needs to know the execution times of the stages of the pipeline. This is true for both homogeneous and heterogeneous platforms. However, for heterogeneous platforms, the tasks in each pipeline stage may take drastically different times when executed on different PUs (*e.g.*, the execution time of a task on a general-purpose core and accelerator may vary by one or two orders of magnitude [42]–[44]). It is necessary to know the mapping and schedule of tasks to PUs to determine a high performance pipeline structure

Unfortunately, the reverse dependency also exists, *i.e.*, mapping and scheduling depends on the results of pipeline stage identification. This is because pipeline stages may execute

concurrently (from different instances of the task graph), in spite of the dependencies that exist across stages in the original program. In other words, the scheduler's view of the pipelined task graph is one where the different sub-graphs corresponding to the pipeline stages are concurrent. Thus, how the task graph is partitioned into stages directly determine which tasks are concurrent, impacting the results of scheduling and mapping.

This is not merely a problem of theoretical interest, as the case studies will illustrate.

Let us consider the application task graph shown in Figure 1(a). Suppose we wish to execute the application on a heterogeneous platform consisting of two PUs, PU_0 and PU_1. Also, suppose that we wish to use two pipeline stages (the argument applies irrespective of the number of pipeline stages). The execution times of each task on each PU are also shown in Figure 1(a). For simplicity, we ignore the cost of communication between PUs in this example (we do consider communication cost in our framework and experiments).

The cyclic dependency between pipeline stage identification, and mapping and scheduling can be broken in two ways:

- **Pipeline, then schedule.** Since we do not know the actual task execution times during pipeline stage identification, we need to make some assumptions. For example, we could use the minimum [1] execution time of each task, *i.e.*, the execution time on the fastest PU, as its execution time for the purpose of pipeline stage identification. This corresponds to an optimistic assumption that all tasks will be executed on their best PUs. Once balanced pipeline stages are created, we schedule the pipelined task graph on the heterogeneous platform.
- **Schedule, then pipeline.** Map and schedule the un-pipelined application on the heterogeneous platform. Use the execution times of the tasks from this schedule to identify pipeline stages.

The two approaches described above effectively break the cyclic dependency by performing one of the steps first, followed by the other. We now evaluate the results of these two approaches for the example in Figure 1(a). We assume a list scheduler [45] optimized for heterogeneous platforms, such as HEFT [46] which schedules each task on the PU where the task will finish earliest, considering the suitability of tasks to PUs, locality of data, and availability of PUs.

Let us first consider the "pipeline, then schedule" approach, where we use the minimum execution time of each task for pipeline stage identification. The execution times of tasks $a-f$ are therefore assumed to be $8, 3, 4, 4, 8, 4$ respectively. The partition $\{a, b, c, d\}$, $\{e, f\}$, shown in Figure 1(b) results in the most balanced pipeline structure, with a critical path of 12 units through each stage. Using the maximum $(16, 9, 13, 7, 20, 12)$ or average execution times of the tasks across the two PUs $(12, 6, 8.5, 5.5, 14, 8)$ still leads to the same pipeline structure shown in Figure 1(b). The schedule for this pipelined task graph on the heterogeneous platform (considering contention and concurrency of the tasks), is shown on the right side of Figure 1(b). The pipeline initiation interval (inverse of throughput) is 23 units.

However, a different pipeline structure and schedule exists, as shown in Figure 1(c), that has an initiation interval of 20 units, which corresponds to 1.15x higher throughput than the pipeline shown in Figure 1(b).

Finally, the "schedule, then pipeline" approach will also result in the same, sub-optimal, pipeline structure as presented in Figure 1(b) as the latency optimized schedule, in this case, executes each task on its preferred PU, resulting in the same pipeline as above.

The arguments and illustration provided in this section clearly underscore the importance of considering the strong inter-dependency between pipeline stage identification and mapping and scheduling of the application onto the heterogeneous platform. This inter-dependency exists, but is far less significant for homogeneous platforms, since execution times do not vary as significantly for different mappings.

In the next section, we present the AHP framework that automates the generation of software pipelines for heterogeneous parallel platforms, while considering these insights.

III. Heterogeneous Pipeline Framework

An overview of the AHP framework appears in Figure 2. AHP takes an annotated C++ program and generates an optimized pipelined program that can be compiled and executed on the heterogeneous platform. AHP starts by analyzing C++ source code with additional AHP compiler directives (Section III-A). The front-end analyzes the program regions that the programmer has annotated as targets for pipelining, and uses the program annotations to extract a task graph for each program region (Section III-B). It also profiles the tasks on the PUs of the heterogeneous platform (Section III-C) to estimate the task execution and inter-task communication times. Next, AHP iteratively performs pipeline stage identification, transforms the task graph to reflect the pipeline structure, and performs mapping and scheduling of the pipelined task graph to the heterogeneous platform (Section III-D). Finally, AHP generates code for the optimized heterogeneous pipeline that uses frameworks such as TBB, Cilk, OpenMP and CUDA (Section III-F). We describe each of these steps in the following sub-sections.

A. Program Annotation

AHP provides a couple of compiler directives (see the grammar in Figure 3(a) and an example in Figure 3(b)) to allow a programmer to specify program regions that are targets for pipelining. The C++ source code fed to AHP consists of five different types of code sections.

1) *Regular code.* Regular C++ code that is not marked by AHP specific compiler directives.
2) *Pipeline sections.* These represent program regions that can be executed in a pipelined manner and should be further analyzed and optimized. Each pipeline section must be enclosed in a while-loop whose body is in turn expressed as tasks. The termination condition of this loop indicates that all data have been processed.
3) *Tasks.* Each task consists of a single code block that may execute on one of the PUs in the heterogeneous platform. Tasks are also annotated with input/output specification that indicates variables that are read (`in`), written (`out`), or read and written (`inout`) within the task. Tasks can be constrained to execute on specific PU

[1] Alternatively the maximum or average could also be used.

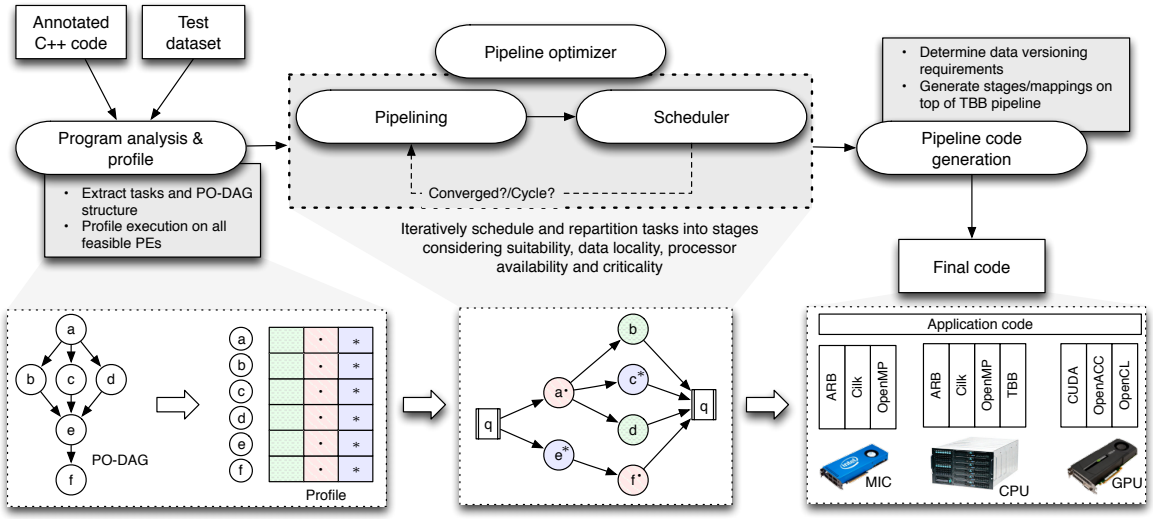


Fig. 2. Overview of the AHP framework.

types, for example, a task requiring access to the disk could be constrained to execute on the host CPU. The programmer can indicate that a task is stateless, *i.e.,* a pure function. For stateless tasks, multiple instances may be created to process different input data.

4) *PU-specific function variants.* Function variants allow the specification of specialized implementations of a task for a specific PU-type. They are useful when (i) different algorithms are utilized for different PU types, or (ii) when PU-specific task implementations cannot be easily generated from a common specification. In our work, we have built upon Intel's compiler that generates code for the MIC [3], while using hand-written CUDA variants for GPU code. AHP could be extended to use existing frameworks such as HMPP [47], OpenACC [14], OpenMPC [15] or OpenCL [10] to generate a larger range of PU-specific implementations of tasks automatically.
5) *Delimited functions.* These functions are delimited, using start and end directives, to indicate that they should be analyzed and optimized for multiple PUs. Each function called within the pipeline section needs to be defined within a delimited section or be specified as a function variant.

The information contained in the AHP compiler directives are similar those required by other heterogeneous programming frameworks [14], [41]. Our proposal of program annotations is intended to identify the information required from the programmer and to illustrate the programming effort, but not as a competing alternative to other standards. We believe that it should be possible to incorporate the AHP directives as minor extensions of other standards.

B. Task Graph Extraction

AHP extracts a task-graph for each pipeline section from the annotated C++ code. The extracted task graph is called a Parallel Operator Directed Acyclic Graph (PO-DAG), since it is a DAG and the tasks may internally exploit data- and instruction-level parallelism (hence, they are parallel operators). The PO-DAG for each pipeline section is constructed using the DAG consistency model [48]. This enforces the data-flow semantics of the original program execution order, while allowing greater scheduling flexibility and out-of-order execution.

A PO-DAG is represented by the tuple of vertices and arcs (V, A). The vertices of the PO-DAG are annotated with the task's code and an arc between two vertices is annotated with the variables being communicated between the tasks.

C. Profiling

AHP needs to determine the execution time of each extracted task on each PU in the heterogeneous platform. As with all profiling-based approaches, we require the profiling data set to be representative. In AHP's case this means that the relative execution times of tasks on PUs using the profiling input are reflective of the relative execution times during productive runs.

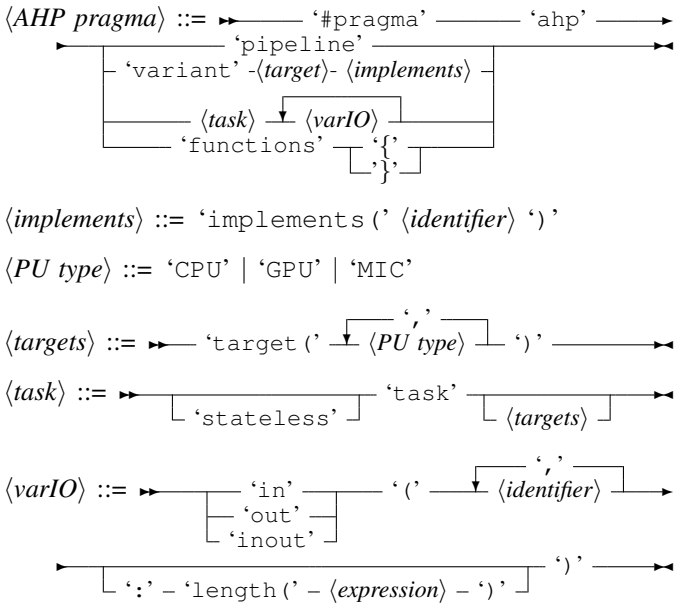
During profiling, the PO-DAG, (V, A), gets extended to the profiled PO-DAG, $(V, A, t_{\text{exec}}, t_{\text{comm}})$, where $t_{\text{exec}} : V \rightarrow \mathbb{R}^P$ denotes the expected execution time of each task on each of the P PUs, and $t_{\text{comm}} : A \rightarrow \mathbb{R}^{P \times P}$ denotes the communication time required to move the data associated with the arc, from one PU to another.

Each task may be executed multiple times during a profiling run, and the framework executes multiple profiling runs. The resultant execution times are recorded and used to compute the average execution times in t_{exec}. The profiling runs are also used to compute the average volume of data communicated on each arc in the PO-DAG. A data transfer model is derived by creating micro-benchmarks that transfers different size data between each pair of PUs. The average data volume is combined with the data transfer model to compute t_{comm}.

D. Pipeline Optimization

The pipeline optimization process consists of two key steps, pipeline stage identification (henceforth called stage identification) and scheduling and mapping (henceforth called scheduling) that are iteratively executed to identify a good pipeline structure.

The high-level description of the procedure for pipeline optimization appears in Figure 4. The inputs are the profiled PO-DAG and a specified number of pipeline stages (the procedure can be iterated to explore different numbers of pipeline stages).


⟨AHP pragma⟩ ::= '#pragma' 'ahp'
'pipeline'
'variant' -⟨target⟩- ⟨implements⟩
⟨task⟩ ⟨varIO⟩
'functions' '{' '}'
⟨implements⟩ ::= 'implements(' ⟨identifier⟩ ')'
⟨PU type⟩ ::= 'CPU' | 'GPU' | 'MIC'
⟨targets⟩ ::= 'target(' ',' ⟨PU type⟩ ')'
⟨task⟩ ::= 'stateless' 'task' ⟨targets⟩
⟨varIO⟩ ::= 'in' 'out' 'inout' '(' ',' ⟨identifier⟩
':' - 'length(' - ⟨expression⟩ - ')' ')'

(a) Grammar for AHP compiler directives.

```
#include <iostream>
#pragma ahp functions {
int scale(double sum, double step)
{
   return step * sum;
}
#pragma ahp functions }
// Note: Just to illustrate variant mechanism, not required.
#pragma ahp variant target(MIC) implements(scale)
int scaleVariant(int onMIC, double sum, double step)
{
   int local_pi;
   #pragma offload target(MIC: onMIC) in(sum)
       ↪out(local_pi)
   local_pi = step * sum;
   return local_pi;
}
int main()
{
   int num_steps = 0;
   const int N = 200;
   float pi, sum;
   #pragma ahp pipeline
   while ((num_steps += 5) < N) {
      // Compute π for num_steps iterations
      #pragma ahp task in(num_steps) out(sum, step)
      {
         step = 1.0/(double)num_steps;
         sum = 0.0f;
         #pragma omp parallel for reduction(+:sum)
         for (int i=0; i < num_steps; i++){
            float x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
         }
      }
      #pragma ahp task in(sum, step) out(pi)
      pi=scale(sum, step);
      #pragma ahp task target(CPU) in(num_steps, pi)
      printf("Steps: %d, pi: %f\n",num_steps,pi);
   }
}
```

(b) Small, complete, program illustrating AHP compiler directive usages.

Fig. 3. AHP program annotations.

input : Profiled PO-DAG $G = (V, A, t_{\text{exec}}, t_{\text{comm}})$ and number of pipeline stages S
output: Optimized pipeline partitioning $\bar{\mathcal{P}}$

create dummy source and sink node;
// Create pipelining graph $G' = (V, A, d, w)$
foreach $v \in V$ **do** $d(v) \leftarrow \min t_{\text{exec}}(v)$;
foreach $a \in A$ **do**
  if $a \in$ outgoing(source) **then** $w(a) \leftarrow S$;
  else $w(a) \leftarrow 0$
end
// Iteratively pipeline and schedule
$\mathcal{P} \leftarrow (V, \emptyset, \ldots, \emptyset)$;
while $\mathcal{P}$ *unseen* **do**
  $\mathcal{P} \leftarrow$ stageIdentification$(G', \mathcal{P})$;
  $G'' \leftarrow$ throughputTransformation$(G, \mathcal{P})$;
  $(G', t) \leftarrow$ schedule$(G'', \mathcal{P})$;
  if *best execution time seen* **then** $\bar{\mathcal{P}} \leftarrow \mathcal{P}$;
end
return $\bar{\mathcal{P}}$

Fig. 4. AHP pipeline optimization algorithm.

As explained in previous sections, stage identification and scheduling are mutually interdependent processes: for stage identification, the execution times and communication costs are required which are in turn determined by the schedule, while the scheduler requires the pipelined structure to know which tasks can concurrently execute. AHP accounts for this cyclic dependency by creating a feedback between these steps and iteratively refining the solution. Specifically, the pipeline optimization procedure in Figure 4 contains a while loop that iteratively performs three steps namely, stage identification, throughput transformation, and scheduling, which we describe in further detail below.

1) Stage identification: Stage identification is the process of determining a good pipeline structure given execution times for the tasks in the PO-DAG. Stages identification is reformulated as the retiming problem [49]. Retiming was originally proposed as a technique for moving registers in a digital circuit to improve performance while preserving its I/O behavior. We recast the problem of stage identification as a retiming problem, by considering the PO-DAG to be a circuit graph, wherein tasks are circuit elements (gates) with delays, and queues are registers. In effect, moving registers to minimize the longest combinational path is equivalent to creating pipeline stages such that their critical paths are balanced.

We next discuss the pipelining algorithm (Figure 5), using terminology specific to our context where appropriate. The PO-DAG can be viewed as a circuit graph used for retiming. For this purpose, we define the propagation delay, d, for each vertex, and queue count, w, for each arc. The queue count of an arc corresponds to the number of queues on that arc, while the propagation delay represents the execution time of the task.

As explained in Figure 4, d is initialized to the minimum execution time of a task across all PUs. A dummy source vertex is added to the PO-DAG, and S queues, where S is the number of stages, are added to each of its outgoing arcs. In other words, w is initialized to S for arcs from the source vertex and 0 for other arcs. The values of d and w are iteratively refined by the pipeline optimization procedure.

The definitions of w and d are extended to paths as

follows [49]. For any path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} v_k$

$$w(p) \equiv \sum_{i=0}^{k-1} w(e_i) \quad \text{and} \quad d(p) \equiv \sum_{i=0}^{k} d(v_i). \tag{1}$$

This leads to two central quantities in solving the retiming problem, namely,

$$W(u,v) := \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, \tag{2}$$

$$D(u,v) := \max\{d(p) : u \overset{p}{\rightsquigarrow} v \ \wedge \ w(p) = W(u,v)\}. \tag{3}$$

For $u, v \in V$, $W(u,v)$ corresponds to the minimum number of queues on any path between u and v in the graph, and $D(u,v)$ corresponds to the maximum total propagation delay (in the terminology of retiming) on any path between u and v that has the minimum number of queues ($W(u,v)$).

These two values are computed first during the pipelining algorithm, after which a binary search over the range of D is performed to find the optimal pipeline structure that minimizes the longest path across all stages (the while loop in Figure 5).

For a given element c in D, a system of linear inequalities is generated and solved to determine whether a retiming is feasible such that any path that does not contain a queue is of length less than or equal to c. If a solution exists, the values of the unknowns (r) are used to update the positions of the queues, and the pipeline stages are extracted from queue positions. To deduce the pipeline stages, the function `findPartion()` performs a breadth-first traversal of the PO-DAG and considers a stage as the section of a PO-DAG between successive frontiers of queues (illustrated in Figure 6(a)).

If all the tasks in a stage are stateless, then the stage is stateless and can be executed concurrently with itself. Such a stage is called a parallel stage. If not, we constrain a stage during execution such that no two instances of it can be in flight concurrently.

The output of this process is a task to stage mapping represented as an S-tuple of sets of tasks. We refer to this mapping as a partitioning and denote it by $\mathcal{P}$.

2) Throughput transformation: We need to transform the PO-DAG to take the pipeline structure into account before scheduling and mapping tasks to PUs. Several heterogeneous task schedulers have been developed for unpipelined applications that optimize for latency [11], [12], [21], [46]. We use such schedulers in the AHP framework by transforming the PO-DAG such that a latency optimized schedule of the transformed PO-DAG corresponds to a throughput optimized schedule of the pipelined PO-DAG.

This transformation consists of creating a new PO-DAG wherein the subgraphs corresponding to the pipeline stages of the original PO-DAG, are independent and can be executed in parallel (see Figure 6(b) for an illustration).

3) Scheduling: Given the transformed PO-DAG, the objective is to schedule the tasks to PUs such that the execution time (latency) is minimized. Scheduling also involves mapping the tasks to specific PUs. Heterogeneous scheduling is an active area of research, and several schedulers have been proposed [11], [12], [21], [46]. In AHP we use MDR [21]. However, any other heterogeneous scheduler can be used as part of the AHP framework.

input : $G' = (V, A, d, w)$
output: Partition $\mathcal{P}$

foreach $u \in V$ **do**
  foreach $a \in$ `outgoing`(u) **do**
    weight a with $(w(a), -d(u))$;
  end
end
compute all pair shortest-path in G_{wd};
foreach $u, v \in V$ **do**
  $(x, y) \leftarrow$ shortest path between u and v;
  $W(u,v) \leftarrow x$;
  $D(u,v) \leftarrow d(v) - y$;
end
RD $\leftarrow$ sort elements in the range of D;
low $\leftarrow 1$;
high $\leftarrow$ `size`(RD);
`// Find minimum achievable period`
while low $\leq$ high **do**
  mid $\leftarrow$ low $+ \lfloor$(high $-$ low)$/2 \rfloor$;
  $c \leftarrow$ RD[mid];
  if $\exists\ R : V \to \mathbb{R},\ r : V \to \mathbb{Z}$

$$\begin{aligned} \textit{s.t. } & r(v) - R(v) \le d(v)/c && ,\forall\, v \in V \\ & R(v) - r(v) \le 1 && ,\forall\, v \in V \\ & r(u) - r(v) \le w(a) && ,\forall\, a = (u,v) \in A \\ & R(u) - R(v) \le w(a) - \frac{d(v)}{c} && ,\forall\, a = (u,v) \in A \end{aligned}$$

  then
    high $\leftarrow$ mid $- 1$;
    $\bar{r} \leftarrow r$;
  end
  else low $\leftarrow$ mid $+ 1$;
end
`// Adjust queue count`
foreach $a = (u,v) \in A$ **do** $w(a) \leftarrow w(a) + \bar{r}(v) - \bar{r}(u)$;
$\mathcal{P} \leftarrow$ `findPartition`(G');
return $\mathcal{P}$;

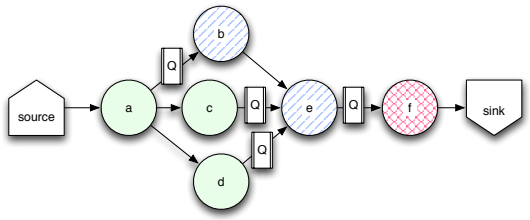
Fig. 5. Pipelining algorithm.

The throughput transformed graph is scheduled to produced a task to PU mapping as well as revising the vertex delays, *i.e.,* the values of d, in the retiming graph. The delay is changed to reflect the total time taken from when the task was ready to execute until it completed, *i.e.,* the time waiting for input data to arrive, the selected PU to become available, as well as the execution time on the mapped PU are considered.

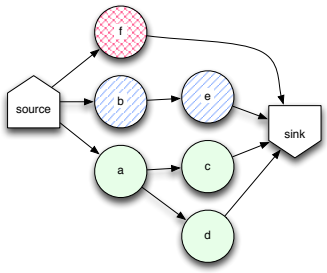
AHP iterates the steps of stage identification, throughput transformation, and scheduling. For any iterative algorithm there is a question of convergence. In AHP, convergence is enforced by detecting a re-occurring partition. If a previously seen partition re-occurs, since the algorithms used are deterministic, either the procedure has converged or a cycle has occurred. In either case, the algorithm returns the best partition seen thus far.

We continue the example from Figure 1 to show how the iterative approach works in Figure 7. In this figure we show how the propagation delay, d, task to PU, and task to stage mapping changes as we iteratively pipeline and schedule. The first iteration produces the pipeline structure and schedule of Pipeline 1 (Figure 1(b)), and the next iteration produces Pipeline 2 (Figure 1(c)), after which the algorithm converges.

The schedule produced by AHP is static, even for tasks in parallel stages for which multiple instances could be in flight simultaneously. Instead of static scheduling for parallel stages,



(a) Stages identified using BFS.



(b) Throughput transformation of G'.

Fig. 6. Using the inserted queues to identify task to stage mappings. By performing a BFS from the source we identify $\mathcal{P} = (\{a, c, d\}, \{b, e\}, \{f\})$. From the identified stages we create the throughput graph by considering the subgraphs induced by the sets of the partition to be independent.

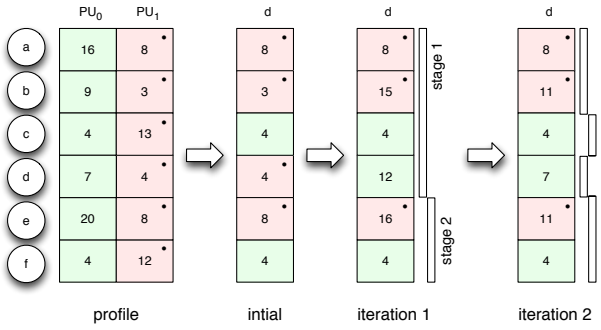


Fig. 7. Illustration of the iterative stage identification and scheduling for the example from Figure 1. Notice how d changes between iterations as availability and communication costs are taken into account.

one could also unroll the loop and create a cyclic schedule using the procedure described.

E. Search Complexity

If we have $|V|$ tasks, S pipeline stages and P PUs, then there are $O(S^{|V|})$ possible task to stage mappings, and $O(P^{|V|})$ possible schedules. The combined search space of possible pipelined realizations is therefore $O(S^{|V|}P^{|V|})$.

Using retiming and scheduling as a search heuristic allows us to efficiently navigate this search space. Retiming is known to have a worst-case complexity of $O(|V|^3 \lg |V|)$, and the list-based scheduler used is $O(|V|^3)$, therefore the running time of our stage identification is $O(|V|^3 \lg |V|)$ per iteration. Although it is difficult to analytically bound the number of iterations of the overall AHP pipeline optimization algorithm (Figure 4), in our experiments we found that the algorithm converged within 4 iterations in all cases. It is always possible to fix the maximum number of iterations.

In practice, in our case studies, we found that the runtime of AHP was dominated by the profiling phase, and the pipeline optimization phase required at most 5 minutes. AHP is a source-to-source compiler framework so the only overhead incurred at run-time is due to the runtime framework used to orchestrate the execution of pipeline stages, maintain inter-stage queues, and perform the data versioning.

F. Final Code Generation

The final step in AHP is to generate the production code for the program using the pipeline structure and schedule determined during pipeline optimization. The code generation step determines the data communication and versioning requirements of each pipeline stage, after which, AHP generates code for the pipeline structure using the pipeline construct of Intel's TBB [25]. AHP determines what data needs to be passed between and through stages. Furthermore, different versions of a data structure will be in flight at the same time and the generated code should manage the versioning correctly and efficiently.

1) Communication across stages: The data communicated from stage i to stage $(i + 1)$ corresponds to data produced in stages 1 to i and consumed in stages $i + 1$ to S (as the TBB pipeline is a linear pipeline). The output of this analysis helps generate the data structures that are populated and manipulated in stages, and passed between pipeline stages. At this point the task-to-PU mapping and communication pattern has been fixed, allowing for the insertion of calls to the explicit data transfer APIs that are required to keep the data coherent.

The two primary approaches to keeping memory coherent between the PUs: distributed shared memory (such as, Intel's MYO [50] and GMAC [51]) and message passing-like models (such as CUDA's explicit memory copy calls and Intel MIC's offload model). The current implementation of AHP uses explicit data transfer calls to maintain memory consistency based on the DAG consistency model. Data structures requiring deep copies (such as trees) are serialized and deserialized using an approach similar to that of the Boost serialization library. However, the distributed shared memory approach may also be used with AHP.

2) Data versioning: A pipelined program has multiple versions of the same data structure in flight. TBB provides the pipeline construct with which to realize pipeline communication, but the data versioning is left to the programmer. AHP determines the data versioning needs and inserts the appropriate calls to the supporting functions: AHP inserts code to duplicate (allocate and copy), merge (recombine results as well as free unnecessary copies) and free data structures.

The number of versions of a data structure is bound by the maximum of the number of pipeline tokens [25] and the number of stages in the pipeline. AHP avoids unnecessary allocation and deallocation by using a circular buffer. This buffer also tracks the last instance of a variable that flowed through the pipeline, useful for assigning the value of live-out variables and reclaiming memory used for versioning.

The optimized pipelined program is generated once the data communication and versioning needs have been determined. The generated program needs to be compiled/linked with PU-specific frameworks (AHP presently targets Intel's MIC offload framework and NVIDIA's CUDA) before it is executed on the target heterogeneous platform.

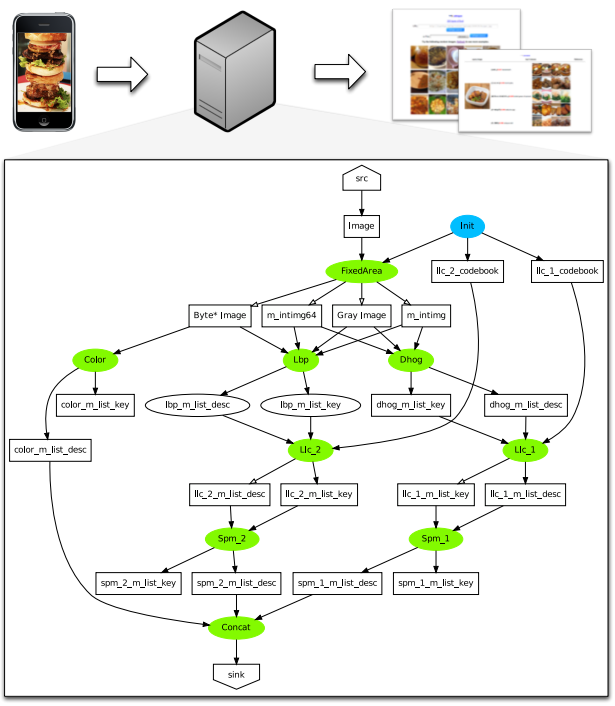


Fig. 8. The feature extraction component of the image classification application.

IV. Evaluation

A. Implementation

We have implemented the AHP framework in C++ using the ROSE open source compiler infrastructure, LEMON [52], GLPK [53], and Intel TBB [25]. We use Intel's ICPC and NVIDIA's NVCC as back end compilers. AHP is a source-to-source compiler but it needs to internally build and execute the program on the target platform for the profiling step.

The AHP compiler, ahp_compile, operates in one of two modes, selected using command-line flags, it either generates the profiling versions or generates the production code using the output from the stage identification program ahp_optimize. Internally ahp_compile uses the back end compilers (icpc or nvcc) to generate the compiled profiling versions. The AHP compiler is called by the top-level AHP command which follows an approach similar to the Intel Software Autotuning Tool [54] to automate the pipeline optimization process.

In our current implementation, the number of stages and the pipeline tokens are given as inputs to AHP . However, these could also be considered as part of the search space for creating an optimized heterogeneous pipeline.

B. Case Studies

We evaluate AHP using three commercial applications that we describe below. In each of these applications, the program regions annotated as pipeline sections are responsible for nearly 100% of the steady state execution time.

1) Image classification: This application performs image classification (IC) as part of an image based search server for mobile devices. Users submit images as search queries, and the server returns images that are similar to the query image. The image classification program that we consider extracts features from the query images, uses machine learning techniques to classify the query into pre-specified categories (*e.g.*, different types of food), and returns results based on the features and categories from the classification. The entire application consists of 153,871 lines of code, of which the most computational intensive part is the feature extraction.

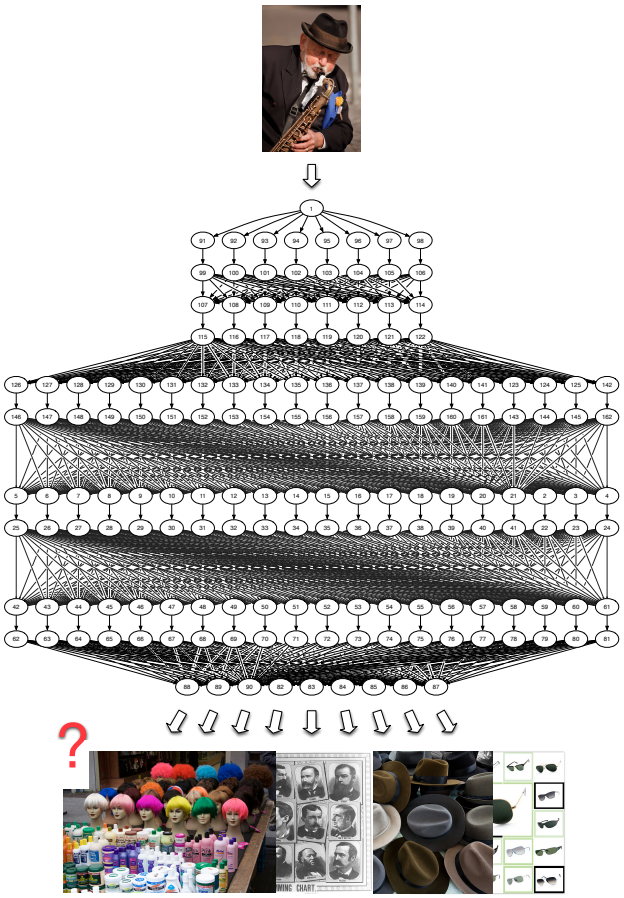


Fig. 9. PO-DAG structure of the pipeline section of the CNN application.

As an illustration of the programmer annotation effort required for AHP, we contrast the number of lines of code with that of a manually written pipeline using TBB [25]. The AHP specification contained a pipeline section that spanned 43 lines of code, of which only 11 lines were AHP compiler directives. In contrast, manually specifying the pipeline using TBB required 281 lines of code.

2) Convolutional neural network: Convolutional neural networks (CNNs) [55] are a special class of multi-layer neural networks used in a wide range of computer vision applications [56] including handwriting recognition, automotive safety, video surveillance, and face detection [57].

In this case study we consider an application of CNNs to object detection in images. The CNN constructed (see Figure 9) consists of 12 layers with varying interconnectivity across layers, ranging from one-to-one to one-to-all.

3) Supervised Semantic Indexing: We consider the dominant computational kernel of the Supervised Semantic Indexing [58] approach to document retrieval. In this application, a document, represented using a term frequency-inverse document frequency [59] vector, is mapped to a latent concept space [60] and matched to the documents in a document library or database

that are most similar to the input document.

Each of the tasks in this application is data parallel but the similarity computation is dominant, accounting for 85% of runtime. The similarity computation is stateless and therefore multiple instances of it could be executing concurrently.

C. Heterogeneous Platforms

We evaluate AHP on two different heterogeneous parallel platforms. The first system is a server that contains dual quad-core Xeon E5620 CPUs (total of eight cores) and a pre-production Intel MIC (Knights Corner) accelerator. The second system consists of dual quad-core Intel Xeon E5520 CPUs and an NVIDIA Tesla C1060 GPU.

D. Results

In this section, we present results that show the performance improvement achieved by the pipelined implementations generated by AHP with suitable baselines. We also use AHP to explore implementations with different number of pipeline stages (without any source code changes).

1) Throughput improvements: We compare the throughput of three different versions of each application: 1) A heterogeneous implementation that uses both data and task parallelism (denoted as *baseline*), 2) A *pipelined baseline*, where we first schedule the tasks to PUs optimally and then pipeline this schedule, and 3) the implementation generated by the AHP framework. The pipelined baseline corresponds to the "schedule, then pipeline" approach described in Section II, the alternative "pipeline, then schedule" approach described in Section II is the first iteration in AHP and therefore the final AHP pipeline is always superior (up to 1.37x). All three versions exploit data and task parallelism as well as heterogeneity.

In the case studies, we used hand coded CUDA kernels and vendor provided libraries for the GPU, and autogenerated code and vendor provided libraries for the MIC. The results are reported on separate graphs, and normalized to the unpipelined baseline version for each system.

In Figure 10(a) and 10(b), we report the throughput on the Xeon+MIC and Xeon+GPU systems normalized to the baseline version on the same system. Pipelining using AHP always increased the throughput (1.37x on average across the systems and up to 1.53x) over the baseline and pipelined baseline.

From the results, we notice the following:

- The pipeline baseline performs similar to AHP when 1) the performance differential between PUs for a task is small, *i.e.,* the execution times, including communication, on different PUs are almost the same, or 2) the performance differential is large, *i.e.,* the execution times vary greatly. Where "small" is within 5% of the best execution time, and "large" is a function of the proportion of time spent in a task and number of tasks.
  In the first case we are nearly in the homogeneous case for which the pipelined baseline is optimal. While in the second case, the size of the difference makes the choice of PU obvious (*i.e.,* fixes the mapping of tasks to PUs) and hence the mapping of task to PU produced in the unpipelined schedule remains optimal even after pipelining.

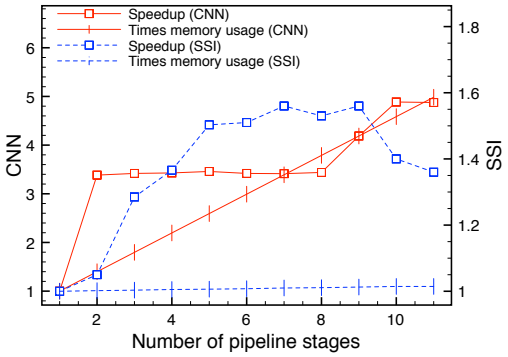


Fig. 11. Comparing number of stages versus speedup and steady-state memory usage.

- For the IC application on the Xeon+MIC, the best mapping resulted when one schedules a task on a PU which is not the best for that task locally. But this task has the same preferred PU as another task, which is more critical (dominant). In the unpipelined application, due to the inter-task dependencies these tasks can never be scheduled concurrently, but they do become concurrent in the pipelined case. This reflects our assertion that pipelining changes the contention profile, which in turn impacts the mapping and scheduling decisions.
- The pipelined baseline for the CNN application on the Xeon+GPU system does not achieve any performance improvement over the unpipelined baseline. This is because, if the original unpipelined PO-DAG is scheduled, GPU-only execution provides the best performance which is identical to the pipelined mapping. Since the Tesla GPU does not have the concurrent kernel execution capability of the newer Fermi [61] platform, it serializes all GPU tasks even in the pipelined execution.

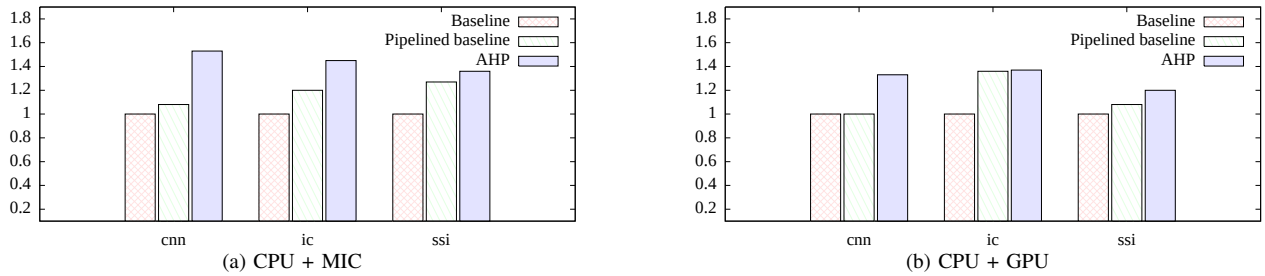
In summary, we conclude that the improvements achieved by AHP over the two baselines establishes the additional benefit from utilizing pipeline parallelism (over and above task and data parallelism) on heterogeneous platforms, as well as the necessity of considering the interaction between pipeline stage identification and scheduling/mapping.

2) Varying the number of stages: Since the program specification in AHP is independent of the number of stages, it is straightforward to generate a range of pipelined implementations by varying this parameter. A tradeoff between performance and memory arises when we vary the number of pipeline stages. For example, a programmer may ask the question: Can we make each task a stage and have as many stages as needed? There are two reasons to minimize the number of stages, the first of which is relevant even for homogeneous pipelines.

Multiple pipelined implementations, with varying numbers of stages, were generated and executed on a homogeneous multi-core system with two quad-core Xeons.

Figure 11 shows the speedup obtained and increase in memory footprint (normalized to the unpipelined baseline) with increasing numbers of pipeline stages, for the SSI and CNN applications. From the graph, we notice that CNN and SSI have different trade-offs that are better understood by analyzing the steady-state memory consumption.

For correct behavior, one only needs to make copies of



(a) CPU + MIC

(b) CPU + GPU

Fig. 10. Execution results of AHP on experimental systems. We compare the throughput of 1) the latency optimized unpipelined execution, 2) pipeline constructed using min-mapping to homogeneous problem, and 3) AHP normalized to the latency optimized throughput on each system.

the variables written to in a pipeline. Since AHP uses a circular buffer, we can express the steady-state memory usage algebraically as $A \cdot (S + 1) + B$, where S is the number of pipeline stages, A the size (memory footprint) of variables that are both read and written, and B the size read-only variables.

- For SSI $A/B \approx 0.002$ and hence memory usage does not greatly increase with the number of stages. However, performance does peak at 7 pipeline stages.
- For CNN $A/B \approx 1.96$, therefore the memory usage increase is comparable to the speedup as we increase the number of pipeline stages. At 7 stages, the memory increase and speedup are roughly equal. However, the performance from 2 stages is as good and uses 2.4X less memory. If we care only about performance, then 10 stages gives the best performance.

While the tradeoff engendered by varying the number of stages is interesting for homogeneous platforms, for heterogeneous platforms there is an added twist: the drivers. The accelerator's driver processes calls made to the accelerator. Either the driver or the programmer is responsible for ensuring reasonable behavior when multiple threads simultaneously call it. This introduces additional, sometimes implicit, communication and synchronization between threads that should be independent. This further limiting factor encourages using as few pipeline stages as possible.

In summary, we believe that our results underscore the utility of the AHP framework in improving performance on heterogeneous platforms with minimal programmer effort.

V. Related work

Pipelines are a fundamental parallel programming pattern [22], [62]–[64] and an approach to structured parallel programming [65]. As such, specialized programming languages [66] as well as language extensions, frameworks and libraries [25], [33]–[36] have been developed to ease the task of constructing software pipelines. These previous approaches target homogeneous systems, but do not consider the key insight that is exploited in AHP.

Creating a heterogeneous pipeline, as AHP does, and creating a pipeline for an accelerator [37]–[39] are two separate problems. In the latter mentioned efforts, all stages of the pipeline are executed on the (homogeneous) cores of the accelerator, and the host CPU is only used for orchestration, leaving its computational capability unutilized. Conceptually, this problem is identical to the problem of pipelining on homogeneous multi-cores.

Thies *et al.* [29] propose a practical approach to exploiting pipeline parallelism in C programs on a homogeneous architecture. The programmer annotates stages using annotations, which they use to extract data flow information, and generate thread and communication management code. It is up to the programmer to identify and balance stages.

Decoupled Software Pipelining (DSWP) [30]–[32] is a compiler parallelization technique that extracts pipeline parallelism from a sequential program. DSWP is proposed for homogeneous architectures and targets fine-grained parallelism; extra architectural support is proposed to enable exploiting finer-grained parallelism.

In feedback-directed pipeline parallelism [67] the question posed is how to best allocate (homogeneous) resources to pipeline stages given a fixed pipeline. AHP instead follows a hierarchical approach whereby the intra-task parallelism is exposed to the underlying PU-specific schedulers, while AHP manages the pipeline parallelism across PUs.

Sanchez *et al.* [68] proposed a dynamic scheduler for pipeline-parallel programs where the task execution times are not predictable.

Stream programming [6], [69], [70], a programming model in which programs are specified using kernels operating on streams of data, is naturally suited to pipelining due to the producer-consumer relationships across kernels [23]. Compiler and runtime frameworks have been developed for stream programming that automatically map the kernels onto homogeneous processing elements [37], [71]–[73].

In summary, none of the previous approaches automatically generate heterogeneous pipelines, taking into consideration the specific challenges and key insights expounded by AHP.

VI. Conclusion

We addressed the problem of automatic software pipelining for heterogeneous platforms. Identifying the inter-dependency between stage identification and scheduling as the key conceptual challenge for heterogeneous pipelining, for which we provide a solution. We presented AHP, a framework requiring minimal programmer effort to automatically generate an optimized heterogeneous pipelined implementation of an annotated C++ application. We demonstrated the benefits of AHP through three case studies on two heterogeneous platforms.

Acknowledgment: This work was supported in part by the National Science Foundation under Grant No. 0916817, and in part by NEC Laboratories America.

REFERENCES

[1] "General-purpose computation on graphics hardware." [Online]. Available: http://gpgpu.org/

[2] IBM, "The Cell project." [Online]. Available: http://www.research.ibm.com/cell/

[3] Intel, "Many Integrated Core (MIC) Architecture." [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

[4] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33 –38, Jul. 2008.

[5] NVIDIA, "CUDA Zone." [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[6] I. Buck, "Brook: A streaming programming language," 2001. [Online]. Available: http://www.cs.washington.edu/research/vlsi/bibliography/Streams/brook_v.02.pdf

[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[8] "Amd app acceleration." [Online]. Available: http://www.amd.com/stream

[9] "Accelerator." [Online]. Available: http://research.microsoft.com/en-us/projects/accelerator/

[10] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, pp. 66–73, 2010.

[11] P. Bellens, J. M. Perez, F. Cabarcas, A. Ramirez, R. M. Badia, and J. Labarta, "CellSs: Scheduling techniques to better exploit memory hierarchy," *Sci. Program.*, vol. 17, pp. 77–95, January 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1507443.1507450

[12] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par '09*, ser. Lecture Notes in Computer Science, vol. 5704. Delft, The Netherlands: Springer, Aug. 2009, pp. 863–874.

[13] CAPS. (2010) HMPP workbench. [Online]. Available: http://www.caps-entreprise.com/hmpp.html

[14] OpenACC, "OpenACC: Directives for accelerators," 2012. [Online]. Available: http://www.openacc-standard.org/

[15] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *SC '10*, Washington, DC, USA, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.36

[16] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[17] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ASPLOS XIII*. New York, NY, USA: ACM, 2008, pp. 287–296.

[18] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *HPDC '08*. New York, NY, USA: ACM, 2008, pp. 197–200.

[19] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO 42*. New York, NY, USA: ACM, 2009, pp. 45–55.

[20] K. Spafford, J. Meredith, and J. Vetter, "Maestro: data orchestration and tuning for OpenCL devices," in *Euro-Par'10*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 275–286.

[21] J. A. Pienaar, A. Raghunathan, and S. Chakradhar, "MDR: performance model driven runtime for heterogeneous parallel platforms," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995933

[22] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*, 1st ed. Addison-Wesley Professional, 2004.

[23] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XII. New York, NY, USA: ACM, 2006, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168877

[24] C. Bienia and K. Li, "Characteristics of workloads using the pipeline programming model," in *Proceedings of the 3rd Workshop on Emerging Applications and Many-core Architecture*, June 2010.

[25] Intel, "Intel threading building blocks (TBB)," 2012. [Online]. Available: http://software.intel.com/en-us/intel-tbb/

[26] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," in *OOPSLA '09*. New York, NY, USA: ACM, 2009, pp. 227–242.

[27] A. Aiken and A. Nicolau, "Perfect pipelining: A new loop parallelization technique," in *Proceedings of the 2nd European Symposium on Programming*, ser. ESOP '88. London, UK, UK: Springer-Verlag, 1988, pp. 221–235. [Online]. Available: http://dl.acm.org/citation.cfm?id=645387.651543

[28] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 318–328. [Online]. Available: http://doi.acm.org/10.1145/53990.54022

[29] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *International Symposium on Microarchitecture*, Chicago, IL, Dec 2007.

[30] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August, "Decoupled software pipelining with the synchronization array," in *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, sept.-3 oct. 2004, pp. 177 – 188.

[31] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, nov. 2005, p. 12 pp.

[32] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 114–123. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356074

[33] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "Parallel programming of general-purpose programs using task-based programming models," in *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, ser. HotPar'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=2001252.2001265

[34] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan, "From patterns to frameworks to parallel programs," *Parallel Comput.*, vol. 28, no. 12, pp. 1663–1683, Dec. 2002. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(02)00190-4

[35] M. Danelutto and P. Teti, "Lithium: A structured parallel programming environment in Java," in *Computational Science ICCS 2002*, ser. Lecture Notes in Computer Science, P. Sloot, A. Hoekstra, C. Tan, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2002, vol. 2330, pp. 844–853, 10.1007/3-540-46080-2_89. [Online]. Available: http://dx.doi.org/10.1007/3-540-46080-2_89

[36] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald, "Using generative design patterns to generate parallel code for a distributed memory environment," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '03. New York, NY, USA: ACM, 2003, pp. 203–215. [Online]. Available: http://doi.acm.org/10.1145/781498.781532

[37] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on GPUs," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 200–209. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.20

[38] X. Wang and T. Zhang, "Fussli: A portable framework for exploiting hybrid task, data and pipeline parallelism on multi-cores," in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 11, oct. 2010, pp. V11–88 –V11–95.

[39] S. Li, S. Yao, H. He, L. Sun, Y. Chen, and Y. Peng, "Extending synchronization constructs in openMP to exploit pipeline parallelism on heterogeneous multi-core," in *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part II*, ser. ICA3PP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 54–63. [Online]. Available: http://dl.acm.org/citation.cfm?id=2075462.2075469

[40] "OpenMP API specification for parallel programming." [Online]. Available: http://openmp.org/wp/

[41] Intel, "Language extension for offloading (LEO)."

[42] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370 – 1380, 2008, ¡ce:title¿General-Purpose Processing using Graphics Processing Units¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731508000932

[43] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *Micro, IEEE*, vol. 28, no. 4, pp. 13 –27, july-aug. 2008.

[44] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser.

ISCA '10. New York, NY, USA: ACM, 2010, pp. 451–460. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816021

[45] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506 –521, May 1996.

[46] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, mar. 2002.

[47] CAPS. (2010) HMPP workbench. [Online]. Available: http://www.caps-entreprise.com/hmpp.html

[48] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "Dag-Consistent Distributed Shared Memory," in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 132–141.

[49] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[50] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, "Programming model for a heterogeneous x86 platform," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 431–440.

[51] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10. New York, NY, USA: ACM, 2010, pp. 347–358.

[52] B. Dezs, A. Jüttner, and P. Kovács, "LEMON - an open source c++ graph template library," *Electron. Notes Theor. Comput. Sci.*, vol. 264, no. 5, pp. 23–45, Jul. 2011.

[53] A. Makhorin, "GLPK (GNU linear programming kit)." [Online]. Available: http://www.gnu.org/software/glpk/glpk.html

[54] C. Luk, "Intel software autotuning tool." [Online]. Available: http://software.intel.com/en-us/articles/intel-software-autotuning-tool/

[55] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, November 1998.

[56] Y. LeCun, K. Kavukvuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Proc. International Symposium on Circuits and Systems (ISCAS'10)*. IEEE, 2010.

[57] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 247–257. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815993

[58] S. Byna, J. Meng, A. Raghunathan, S. Chakradhar, and S. Cadambi, "Best-effort semantic document search on GPUs," in *GPGPU '10*. New York, NY, USA: ACM, 2010, pp. 86–93.

[59] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, Nov. 1975. [Online]. Available: http://doi.acm.org/10.1145/361219.361220

[60] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society of Information Science*, vol. 41, no. 6, pp. 391–407, 1990. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.7546

[61] 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[62] S. Siu, M. D. Simone, D. Goswami, and A. Singh, "Design patterns for parallel programming," in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 1996, pp. 230–240.

[63] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389 – 406, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819104000080

[64] B. L. Massingill, T. G. Mattson, and B. A. Sanders, "Reengineering for parallelism: an entry point into plpp for legacy applications: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 4, pp. 503–529, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.v19:4

[65] M. D. McCool, "Structured parallel programming with deterministic patterns," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, ser. HotPar'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863086.1863091

[66] N. Mahajan, S. Sasidharan, A. Chauhan, and A. Lumsdaine, "Automatically generating coarse grained software pipelining from declaratively specified communication," in *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, Dec. 2011, short paper in the Student Research Symposium.

[67] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed pipeline parallelism," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 147–156. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854296

[68] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *PACT*, 2011, pp. 22–32.

[69] "StreamIt language specification." [Online]. Available: http://cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf

[70] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM, 2004, pp. 777–786.

[71] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA USA, Oct 2002. [Online]. Available: http://groups.csail.mit.edu/commit/papers/02/streamit-asplos.pdf

[72] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi, "Merrimac: Supercomputing with streams," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 35–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050187

[73] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, ser. PACT '06. New York, NY, USA: ACM, 2006, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/1152154.1152164